CS 4820, Summer 2010 Out: July 13. **Due: Friday, July 16, 8:30am**

Homework 3

Reading: Chapter 4, Sections 6.1, 6.2, 6.3, 6.6.

Problem 1. Some security consultants working in the financial domain are currently advising a client who is investigating a potential money-laundering scheme. The investigation thus far has indicated that n suspicious transactions took place in recent days, each involving money transfered into a single account. Unfortunately, the sketchy nature of the evidence to date means that they don't know the identity of the account, the amounts of the transactions, or the exact times at which the transactions took place. What they do know is an *approximate time-stamp* for each transaction; the evidence indicates that transaction i took place at time $t_i \pm e_i$ for some "margin of error" e_i. (In other words, it took place some time between $t_i - e_i$ and $t_i + e_i$.) Note that different transactions may have different margins of error.

In the last day or so, they have come across a bank account that (for other reasons we don't need to go into here) they suspect might be the one involved in the crime. There are n recent *events* involving the account, which took place at times $x_1, x_2, ..., x_n$. To see whether it's plausible that this really is the account they are looking for, they're wondering whether it is possible to associate each of the account's n events with a distinct one of the n suspicious transactions in such a way that, if the account event at time x_i is associated with suspicious transaction that occurred approximately at time t_j, then $|t_j - x_i| \leq e_j$. (In other words, they want to know if the activity on the account lines up with suspicious transactions to within the margin of error; the tricky part here is that they don't know which account event to associate with which suspicious transaction.)

Give an efficient algorithm that takes the given data and decides whether such an association exists. If possible, you should make the running time be at most $O(n^2)$.

Problem 2. After riding quite a bit on hill-optimal paths, you have decided to show off your stamina, and have decided to take part in a strange time-trial competition. The strange part is that you get to choose among various courses you ride along. This is where your strength comes in! Your stamina from limited hill training is good for some time, but it quickly tapers off, and you need to take breaks every once in a while. After carefully studying the pattern of your riding and also the patterns of available courses in the time-trial event, you have decided that you will get the best results by not doing three courses in a row. For any sequence of three consecutive courses, you need to skip at least one. And the trick is that doing different courses gives you different number of points (and let us say you know exactly how many points you get by riding a certain course).

So, here you go! There are n courses that bikers can ride for n consecutive hours. Each course i gives your p_i points. You can ride at most two courses consecutively (and then need to rest for the duration of at least one course, that is, about one hour). Give an algorithm that determines the maximum number of points you can get without ever biking three consecutive courses.

Example: If $n = 5$ with $p_1 = 2$, $p_2 = 3$, $p_3 = 8$, $p_4 = 9$, and $p_5 = 4$, then the optimal solution is to bike course numbers $(1, 3, 4)$ giving total 19 points. Your stamina does allow you to bike $(1, 2, 4, 5)$ (since you are not biking three in a row), but the total number of points you get is equal to 18.

Problem 3. You are now so obsessed with your biking performance, that you have started to analyze the data you gather during your weekly rides (thanks to the sophisticated bike computer you got as a gift from a fellow rider). So, every week, you do a similar route, and calculate the *average speed* during the ride. After consulting with some bike-savy people, you have come to the conclusion that how much you improve is directly related to how long *rising trend* you can find in the *average speed data*. Let us be more formal.

Let us say you have the data for n consecutive weekly rides; each data point is the average speed for that week. So, $s[i]$ gives the average speed for i-th week's ride. A *rising trend* for this sequence of speeds is a subsequence $i_1 < i_2 < i_3 < \cdots < i_k$ such that $s[i_1] < s[i_2] < \cdots < s[i_k]$. Notice that the subsequence does not have to be contiguous.

Now, you are interested in finding the longest rising trend in the weekly-speed data from your rides. For example, for $n = 8$ and speeds equal to $27, 15, 26, 18, 29, 20, 17, 19$, one longest rising trend comes from the subsequence $(2, 3, 5)$ which gives speeds $15 < 26 < 29$.

(3a) Show that the following algorithm does not correctly return the *length* of the longest trend, by giving a counterexample on which it fails to return the correct answer. In your example, give the actual length of the longest rising trend, and say what the algorithm returns.

```
Let $L$ be an array of length $n$, indexed by $[1,2,3...,n]$
for ( firstweek $= 1$ ; firstweek $\leq n$ ; firstweek$++$)
    // compute the length of longest rising trend starting at firstweek
    $L$[firstweek] $= 1$
    $i \leftarrow$ firstweek
    for ( $j = i+1$ ; $j \leq n$ ; $j++$ )
        if ($s[j] > s[i]$)
            $i \leftarrow j$
            $L$[firstweek] $\leftarrow L$[firstweek] $+ 1$
Compute the $\max_{1 \leq i \leq n} L[i]$ and return this value as the length of the
  longest rising trend.
```

(3b) Give an efficient algorithm that takes a sequence of weekly speeds $(s[i] : 1 \leq i \leq n)$, and returns the *length* of the longest rising trend.